Using RDMA Efficiently for Key-Value Services

Anuj Kalia (CMU)
Michael Kaminsky (Intel Labs), David Andersen (CMU)
RDMA

*Remote Direct Memory Access:* A network feature that allows direct access to the memory of a remote computer.
1. Improved understanding of RDMA through micro-benchmarking

2. High-performance key-value system:
   - Throughput: 26 Mops \((2X \text{ higher than others})\)
   - Latency: 5 \(\mu\text{s} \ (2X \text{ lower than others})\)
RDMA intro

Features:

- Ultra-low latency: 1 µs RTT
- Zero copy + CPU bypass

Providers:
InfiniBand, RoCE,…
# RDMA in the datacenter

## 48 port 10 GbE switches

<table>
<thead>
<tr>
<th>Switch</th>
<th>RDMA</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mellanox SX1012</td>
<td>YES</td>
<td>$5,900</td>
</tr>
<tr>
<td>Cisco 5548UP</td>
<td>NO</td>
<td>$8,180</td>
</tr>
<tr>
<td>Juniper EX5440</td>
<td>NO</td>
<td>$7,480</td>
</tr>
</tbody>
</table>
In-memory KV stores

Interface: GET, PUT

Requirements:
- Low latency
- High request rate
RDMA basics

Verbs

RDMA read:
READ(local_buf, size, remote_addr)

RDMA write:
WRITE(local_buf, size, remote_addr)
Life of a WRITE

1: Request descriptor, PIO
2: Payload, DMA read
3: RDMA write request
4: Payload, DMA write
5: RDMA ACK
6: Completion, DMA write

CPU, RAM → RNIC → CPU, RAM
Recent systems

Pilaf [ATC 2013]

FaRM-KV [NSDI 2014]: an example usage of FaRM

Approach: RDMA reads to access remote data structures

Reason: the allure of CPU bypass
The price of CPU bypass

Key-Value stores have an inherent level of indirection.

An index maps a keys to address. Values are stored separately.

At least 2 RDMA reads required:
- $\geq 1$ to fetch address
- $1$ to fetch value

Not true if value is in index
The price of CPU bypass
The price of CPU bypass
The price of CPU bypass
# Our approach

<table>
<thead>
<tr>
<th>Goal</th>
<th>Main ideas</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1: Use a single round trip</td>
<td>Request-reply with server CPU involvement + WRITEs faster than READs</td>
</tr>
<tr>
<td>#2. Increase throughput</td>
<td>Low level verbs optimizations</td>
</tr>
<tr>
<td>#3. Improve scalability</td>
<td>Use datagram transport</td>
</tr>
</tbody>
</table>
#1: Use a single round trip
#1: Use a single round trip

<table>
<thead>
<tr>
<th>Operation</th>
<th>Round Trips</th>
<th>Operations at server’s RNIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ-based GET</td>
<td>2+</td>
<td>2+ RDMA reads</td>
</tr>
<tr>
<td>HERD GET</td>
<td>1</td>
<td>2 RDMA writes</td>
</tr>
</tbody>
</table>

Lower latency  ? High throughput
RDMA WRITEs faster than READs

Setup: Apt Cluster
192 nodes, 56 Gbps IB
RDMA WRITEs faster than READs

Reason: PCIe writes faster than PCIe reads
High-speed request-reply

Request-reply throughput:

Setup: one-to-one client-server communication

Throughput (Mops)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>1</td>
</tr>
<tr>
<td>WRITEs</td>
<td>2</td>
</tr>
<tr>
<td>READs</td>
<td>2</td>
</tr>
</tbody>
</table>

32 byte payloads
#2: Increase throughput

Simple request-reply:

```
Client
CPU,RAM  RNIC

Server
RNIC  CPU,RAM

WRITE #1: Request
Processing
WRITE #2: Response
```
Optimize WRITEs

+ inlining: encapsulate payload in request descriptor (2→1)

+ unreliable: use unreliable transport (-5)

+ unsignaled: don’t ask for request completions (-6)
#2: Increase throughput

Optimized request-reply:

WRITE #1: Request

Processing

WRITE #2: Response
#2: Increase throughput

Setup: one-to-one client-server communication
#3: Improve scalability

![Diagram showing setup with C1, CN, 1, and N connected by arrows and a graph showing throughput (Mops) vs. number of client/server processes.]
#3: Improve scalability

Clients

SRAM

State 1

State 2

State 3

......

State N

|state| > SRAM

C1

C2

C3

CN
#3: Improve scalability

Inbound scalability » outbound because inbound state (□) ≪ outbound (□)

Use datagram for outbound replies

Datagram only supports SEND/RECV. SEND/RECV is slow.

SEND/RECV is slow only at the receiver
Scalable request-reply

- RDMA write, connected
- SEND, datagram

Setup

![Diagram showing scalable request-reply with C1, CN, and S nodes, connecting through RDMA write and SEND, datagram.]
Evaluation

HERD = Request-Reply + MICA [NSDI 2014]

Compare against emulated versions of Pilaf and FaRM-KV

• No datastore

• Focus on maximum performance achievable
Latency vs throughput

48 byte items, GET intensive workload

Latency (microseconds)

Throughput (Mops)

HERD

95th percentile

5th percentile

26 Mops, 5 µs

Low load, 3.4 µs

0

5

10

15

20

25

30

0

4

8

12
Latency vs throughput

48 byte items, GET intensive workload

Latency (microseconds)

Throughput (Mops)

Emulated Pilaf  Emulated FaRM-KV  HERD

95th percentile

5th percentile

12 Mops, 8 µs

26 Mops, 5 µs

Low load, 3.4 µs
Throughput comparison

16 byte keys, 95% GET workload

- Red diamond: Emulated Pilaf
- Green triangle: Emulated FaRM-KV
- Blue triangle: HERD

Throughput comparison chart showing different throughput values for Emulated Pilaf, Emulated FaRM-KV, and HERD based on value size (Bytes) and throughput (Mops). The chart indicates a 2X higher performance for Emulated Pilaf compared to Emulated FaRM-KV and HERD in the 16 byte keys, 95% GET workload scenario.
HERD

• Re-designing RDMA-based KV stores to use a single round trip

  • WRITEs outperform READs

  • Reduce PCIe and InfiniBand transactions

  • Embrace SEND/RECV

• Code is online: https://github.com/efficient/HERD
Throughput comparison

16 byte keys, 95% GET workload

- Emulated Pilaf
- Emulated FaRM-KV
- HERD
- README

Faster than RDMA reads
Throughput comparison

48 byte items

- Emulated Pilaf
- Emulated FaRM-KV
- HERD

Throughput (Mops)

- 5% PUT
- 50% PUT
- 100% PUT